Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading *

Jack Wadden Alexander Lyashevsky§ Sudhanva Gurumurthi† Vilas Sridharan‡ Kevin Skadron

University of Virginia, Charlottesville, Virginia, USA
†AMD Research, Advanced Micro Devices, Inc., Boxborough, MA, USA
§AMD Research, Advanced Micro Devices, Inc., Sunnyvale, CA, USA
‡ RAS Architecture, Advanced Micro Devices, Inc., Boxborough, MA, USA
{wadden,skadron}@virginia.edu
{Alexander.Lyashevsky,Sudhanva.Gurumurthi,Vilas.Sridharan}@amd.com

Abstract

Reliability for general purpose processing on the GPU (GPGPU) is becoming a weak link in the construction of reliable supercomputer systems. Because hardware protection is expensive to develop, requires dedicated on-chip resources, and is not portable across different architectures, the efficiency of software solutions such as redundant multithreading (RMT) must be explored.

This paper presents a real-world design and evaluation of automatic software RMT on GPU hardware. We first describe a compiler pass that automatically converts GPGPU kernels into redundantly threaded versions. We then perform detailed power and performance evaluations of three RMT algorithms, each of which provides fault coverage to a set of structures in the GPU. Using real hardware, we show that compiler-managed software RMT has highly variable costs. We further analyze the individual costs of redundant work scheduling, redundant computation, and inter-thread communication, showing that no single component in general is responsible for high overheads across all applications; instead, certain workload properties tend to cause RMT to perform well or poorly. Finally, we demonstrate the benefit of architectural support for RMT with a specific example of fast, register-level thread communication.

1. Introduction

As data centers and the high-performance computing (HPC) community continue to adopt GPUs as "throughput processors," the graphics-specific nature of these architectures has evolved to support programmability [30]. Alongside new features, greater performance, and lower power, higher reliability is becoming increasingly important in GPU architecture design, especially as we approach the exascale era. An increase in radiation-induced transient faults due to shrinking process technologies and increasing operating frequencies [8, 27], coupled with the increasing node count in supercomputers, has promoted GPU reliability to a first-class design constraint [2].

To protect against transient faults, CPUs and GPUs that run sensitive calculations require error detection and correction.

Structure	Size	Estimated ECC Overhead
Local data share	64 kB	14 kB
Vector register file	256 kB	56 kB
Scalar register file	8 kB	1.75 kB
R/W L1 cache	16 kB	343.75 B

Table 1: Reported sizes of structures in an AMD Graphics Core Next compute unit [4] and estimated costs of SEC-DED ECC assuming cache-line and register granularity protections.

These capabilities typically are provided by hardware. Such hardware support can manifest on large storage structures as parity or error-correction codes (ECC), or on pipeline logic via radiation hardening [19], residue execution [16], and other techniques. However, hardware protection is expensive, and can incur significant area, power and performance overheads. To illustrate the potential cost of hardware protection in a modern GPU, Table 1 shows estimated SEC-DED ECC overheads based on common cache line and register ECC schemes and the reported sizes of memory structures on an AMD Graphics Core Next (GCN) compute unit (CU) [4]. Each CU would need 72kB of ECC, a 21% overhead.

Not only is hardware protection expensive, it is inflexible and not needed for some workloads. While many HPC workloads, such as scientific and financial applications, demand high precision and correctness, rendering applications can be inherently fault-tolerant. Such applications display tens of mega-pixel frames per second, and isolated or transient errors generally are not even observable [26]. GPUs need to serve both communities. Multiple versions of a processor can be designed to satisfy these diverse reliability needs, but this complicates design cycles, and can be prohibitively expensive for the vendor or consumer. Furthermore, users may need the same product to serve diverse workloads (e.g., in virtualized clusters).

The expense and varied requirements for reliability in GPUs motivate the design of GPUs with flexible reliability solutions that provide economical and tunable power, performance, and reliability trade-offs. This motivates the exploration of software reliability on GPUs to provide low-cost, portable, and flexible fault protection, and of compiler solutions to automate the burden of design and implementation of reliability.

*This work was performed while Jack Wadden was a co-op in AMD Research.

978-1-4799-4394-4/14/$31.00

To our knowledge, no prior work has explored automatic software RMT transformations for GPU kernels or the performance trade-offs of GPU RMT with different coverage domains. We focus on detection because it needs to run continuously and is therefore performance critical. The choice of recovery techniques (e.g. checkpoint/ restart or containment domains [7]) is orthogonal. Compiler-managed software GPU RMT automatically provides transient fault detection within the targeted coverage domain and also allows evaluation on existing hardware.

The key contributions of the paper and the findings of our study are:

- We design and develop three RMT algorithms, each of which provides fault coverage for a set of structures in the GPU, and implement them in a production-quality OpenCL™ compiler.
- We carry out detailed performance and power analyses of these algorithms for a set of kernels from the AMD OpenCL SDK benchmark suite on an AMD Radeon™ HD 7790 GPU. We use hardware performance counters and carefully designed experiments that isolate the various overheads of RMT, including redundant work scheduling, redundant computation, and inter-thread communication.
- Our analysis reveals that RMT performance varies over a wide range, from slight speed-ups to large slow-downs. We present several insights into workload and hardware characteristics that determine performance. In general, we find that no single aspect of RMT (e.g., inter-thread communication) fully explains the performance of RMT across all workloads; however, certain workload properties tend to cause RMT to perform well or poorly.
- We find that RMT does not cause an appreciable increase in the average power consumption of the GPU. Therefore, the energy usage of a GPU will be driven mainly by the execution time of RMT-ized kernels.
- We explore the potential benefits of going beyond the OpenCL specification and evaluate the use of an architecture-specific feature to optimize RMT performance, namely register-level swizzling for fast thread-to-thread data exchange. We find that this optimization provides significant performance improvements for several kernels, thus motivating further research into hardware techniques to improve RMT performance on GPUs.

2. Related Work

Redundant multithreading on the CPU is a well-researched area, and automatic solutions have been proposed for both simulated hardware [11, 18, 23, 24, 32] and software [14, 17, 21, 28, 34, 36]. Wang et. al. [34] uses automatic compiler transformations to create fully redundant threads on the CPU, managing both communication and synchronization of operations that exit the SoR. On existing hardware, utilizing a software queue for communication, their technique saw overheads of 4-5x. State-of-the-art CPU redundancy techniques offer impressively low overheads [28, 36]. However, these techniques double node level memory capacity requirements and rely on spare execution resources to execute redundant processes, which may be scarce when running parallel applications on server or HPC systems.

Newer research into reliability solutions for GPUs re-targets the large body of CPU error-detection work to the GPU's unique programming model abstractions and architecture. Simulated hardware extensions have been proposed as low-cost reliability solutions [15, 20, 29]. Software techniques that provide reliability for GPUs generally suffer from large execution overheads [9]. To improve on these performance overheads, state that has a high probability of propagating faults to program output can be selectively protected [35].

An initial exploration of software GPU RMT by Dimitrov et al. [9] showed promise for executing redundant code efficiently on GPUs. The paper implemented full duplication as well as two more sophisticated techniques, R-Scatter and R-Thread, with detection performed on the CPU after kernel execution. These techniques applied to older VLIW architectures and/or were hand-coded, limiting their practical usefulness.

Our techniques differ from prior work in several ways. First, with ECC becoming a standard off-chip DRAM feature, we assume protection in storage and transfer to off-chip resources and target an on-chip protection domain. Second, as GPUs become first-class computing citizens, the model of checking results on the CPU will not be feasible (e.g., as GPUs become capable of directly communicating to I/O). Therefore, we focus on detection on the GPU itself. Third, hardware itself is expensive to implement, inflexible, and necessitates evaluation in GPU simulators that may not reflect proprietary hardware. Therefore, we focus our attention on software-only GPU reliability solutions that can be implemented and evaluated on real hardware.

Finally, selective replication solutions allow for a nonzero transient fault rate within the domain of fault protection, and the level of protection can depend on application-specific behavior. In data centers and large supercomputers, allowing any faults within a "reliable" domain is unacceptable because it prevents guarantees about correctness in the entire system. This is an important, but often overlooked, subtlety; fault detection via RMT enables both processor vendors and designers of software frameworks to reason about and account for unprotected structures. We therefore focus our attention on efficient fully redundant solutions and do not consider probabilistic protection.

3. Background

3.1. Fault Modes

Processors created with traditional manufacturing processes are vulnerable to both permanent (hard) and transient (soft) faults in logic. Permanent faults manifest as stuck-at bits. They can be caused by process variation, thermal stress, or oxide wear-out, and may develop over time as a combination of these phenomena. Permanent faults can be mitigated by using *burn-in*, a technique to eliminate early permanent faults,

before a processor enters its useful lifetime [19]. Periodic stress tests also can be applied during the useful lifetime of hardware to determine if permanent faults have developed [12]. Therefore, we focus on mitigating transient faults.

Transient faults can be caused by crosstalk, voltage violations, and other electromagnetic interference, but are typically associated with the effects of high-energy particles, usually neutrons [8]. Although they do not inject charge, neutrons that strike a chip have the potential to create secondary, ionizing radiation. If enough charge is injected into a transistor's diffusion region to overcome the device's critical charge, it may drive the wrong value temporarily; these temporary upsets in a transistor's state are called single-event upsets (SEUs) or, if more than one transistor is affected, single-event multi-bit upsets (SEMUs) [19]. If an SEU occurs directly within a storage cell on-chip, the incorrect value may persist. If an SEU occurs in combinational logic, it may propagate incorrect values to storage cells. SEUs that are captured and committed to program output are called silent data corruptions (SDC).

3.2. OpenCL Programming Model

This work focuses on implementing RMT in GPU kernels written in OpenCL [30]. OpenCL is an open standard for parallel and heterogeneous computing that targets multicore CPUs and GPUs. In OpenCL, a computation is divided between the host application on the CPU and a kernel program that runs on a parallel accelerator called the device. To launch a kernel, the application developer designates a group of threads or work-items to execute on the device; this collection of executing threads is referred to as the global N-dimensional range, or NDRange. Each work-item in a global NDRange executes the same kernel program, and shares a global address space.

The work-items in a global NDRange are subdivided into work-groups that have the ability to share a local scratchpad memory space. Work-items from one work-group are prohibited from accessing the local memory of another work-group even if it is physically possible on the device architecture. On GPUs, local memory generally has the added advantage of being low-latency relative to global memory. Work-groups can guarantee synchronization among constituent work-items by executing a local barrier instruction. Ordering among work-items in different work-groups is not guaranteed by the OpenCL standard, and therefore must be implemented explicitly if desired.

Within a work-group, each work-item has a private memory space that is inaccessible to other work-items. If executing on single-instruction/multiple-data (SIMD) architectures such as GPUs, work-items are grouped into wavefronts that execute in lockstep on the same SIMD unit. How wavefronts execute on SIMD units is discussed in the following subsection.

3.3. AMD GCN Compute Unit Architecture

While our proposed techniques are generalizable to any architecture, we have implemented and evaluated our designs on GPUs with AMD's GCN CU architecture. AMD GPUs based on the GCN CU architecture provide a certain number of CUs for computation that varies from product to product. Figure 1 shows a high-level block diagram of a single GCN CU. Each CU has four 16-wide SIMD units capable of executing a single 64-wide vector instruction over 4 cycles. Each CU also has its own 64-kB register file, capable of supporting up to 256 64x32-bit vector general-purpose registers (VGPRs). Each CU also has 64-kB of LDS, a low-latency scratchpad memory where OpenCL local memory is allocated.

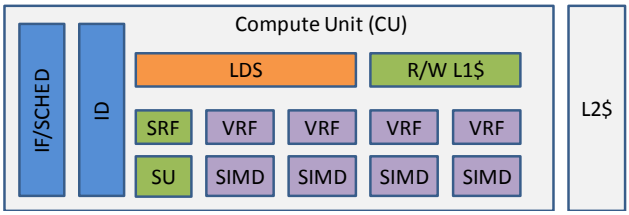

Figure 1: AMD Graphics Core Next compute unit [4]: instruction fetch (IF), scheduler (SCHED), instruction decode (ID), 8-kB scalar register file (SRF), scalar unit (SU), 64-kB vector register file (VRF), SIMD functional units (SIMD), 64-kB local data share (LDS), 16-kB L1 read/write cache (R/W L1$)

When an OpenCL program executes, work-groups are scheduled onto individual CUs. Multiple work-groups can be scheduled onto a single CU if enough LDS and registers are available. Once a work-group(s) is scheduled onto a CU, instructions from wavefronts are issued to SIMD units. The number of wavefronts that can issue instructions on a particular SIMD depends on the presence of available resources. For example, if a wavefront requires 64 VGPRs, only three other wavefronts could potentially issue instructions to that SIMD unit. As many as 10 wavefronts can be scheduled onto a single SIMD unit at any given time.

Each CU also includes one scalar unit (SU) to improve SIMD execution efficiency. When possible, the compiler identifies computations on data common to all threads in a wavefront and executes such instructions on a single data value. Each SU has an 8-kB scalar register file (SRF) that also can affect the number of wavefronts that can execute.

3.4. Redundant Multithreading on the GPU

All methods to detect an unwanted change of state ultimately rely on some sort of redundancy. This can either be encoded versions of state or full duplication.

Fully replicated state is said to be within a sphere of replication (SoR), and is assumed to be protected. Hypothetically, two faults could create simultaneous identical errors in redundant state, but this is considered sufficiently unlikely to be ignored. All values that enter the SoR must be replicated (called input replication) and all redundant values that leave the SoR must be compared (called output comparison) before a single correct copy is allowed to leave [18].

RMT accomplishes replication by running two identical redundant threads–a producer and consumer–on replicated input. Whenever state needs to exit the SoR (e.g., on a store to unreplicated global memory), the producer sends its value to

the consumer for checking before the consumer is allowed to execute the instruction. Hardware structures outside the SoR must be protected via other means.

The OpenCL hierarchical thread programming model allows for thread duplication at different granularities. Naively, entire kernel launches can be duplicated into primary and secondary kernels, and the resulting output can be compared by the host application [9]. If an output mismatch is detected by the host, both redundant copies of the kernel must be re-executed. If we cannot afford full naive duplication, then we must duplicate computation either between OpenCL work-items within a work-group, which we call *Intra-Group RMT*, or across entire OpenCL work-groups, which we call *Inter-Group RMT*.

4. Compiler Implementation

We adapt the automatic compiler transformations of Wang et al. and apply them to the GPU OpenCL programming model. We modify a production-quality OpenCL kernel compiler toolchain [33] to automatically transform OpenCL kernels into RMT programs for error detection. The compiler toolchain consists of three main components. The high-level compiler converts OpenCL kernels into LLVM compiler framework intermediate representation (IR) code [1]. The LLVM layer optimizes and converts the LLVM IR to an architecture-independent GPU intermediate language. A backend-compiler further optimizes and compiles the intermediate language into a architecture-specific GPU program.

We implement our compiler transformation as an LLVM optimization pass in the LLVM layer of the compiler toolchain. This choice gives us enough control to remain general across architectures that adhere to OpenCL specifications and allows us to ensure correct protection semantics. A low-level implementation in the shader compiler would have a high level of control with respect to efficiency and coverage, but would be applicable only to a small set of target architectures. However, a low-level implementation also may allow us to implement fine-grained, architecture-specific improvements. We examine this trade-off in Section 8.

While our OpenCL compiler extension makes automatic transformations to OpenCL kernels, we do not provide automatic transformations for host code. In practice, the host-code modifications necessary to support RMT were small and did not justify the added engineering effort for this research.

5. Evaluation Methodology

Tested Benchmarks: We applied our automatic kernel transformations to 16 kernels from the AMD OpenCL SDK sample suite [5] and verified correct execution by using each application's built-in verification capabilities. When allowed by the application, input sizes were scaled to the maximum size at which all versions of our transformations were able to run. For all benchmarks that allow input scaling, we were able to saturate the test GPUs CUs with work-groups, ensuring a high level of utilization and a fair evaluation.

	SIMD ALU	VRF	LDS	SU	SRF	ID	IF/SCHED	R/W L1$
Intra-Group+LDS	✓	✓	✓					
Intra-Group-LDS	✓	✓						

Table 2: CU structures protected by Intra-Group RMT. Because the LDS is software managed, we can choose to include it or exclude it from the SoR. Other structures shared in SIMD computation are not protected.

Performance and Power Evaluation: All experiments were conducted on a commodity AMD Radeon HD 7790 GPU with 12 CUs. GPU kernel times and performance counters were gathered using AMD's CodeXL^{TM} tool version 1.2 [3]. Kernel runtimes were averaged over 20 runs of each application. Because the GPU's dynamic frequency-scaling algorithms may inconsistently affect the runtimes of our experiments, clock frequencies were fixed at the maximum rated commercial values: 1-GHz core frequency and 1.5-GHz memory frequency.

Power was measured by reading an on-chip power monitor that estimates average ASIC power [10] at an interval of 1ms. These measurements then were averaged over the runtime of the kernel to estimate total average power. Because the reported power estimation is actually an average of instantaneous power calculated using a sliding window, meaningful results could not be gathered from applications with short kernel runtimes.

6. Intra-Group RMT

Intra-Group RMT replicates state and computation by doubling the size of each work-group and then creating redundant work-item pairs within the larger work-group using OpenCL's work-item identification numbers. Because OpenCL allows work-items within a work-group to communicate via local memory, we can use the local data share to implement output comparisons between work-items. Because local memory is software-managed and redundant work-items within a work-group share a local address space, we can choose either to include or exclude its allocation from the SoR. Therefore, we divide Intra-Group RMT into two flavors: *Intra-Group+LDS*, in which LDS allocations are duplicated explicitly, therefore including LDS in the SoR, and *Intra-Group-LDS*, in which LDS allocations are not duplicated, therefore excluding LDS from the SoR. Each SoR, specific kernel modifications, and performance of both Intra-Group RMT flavors are discussed below.

6.1. Sphere of Replication

Table 2 shows the high-level CU structures protected by Intra-Group RMT. Because OpenCL allocates separate registers for each redundant work-item, the entire vector register file (VRF) is protected for both Intra-Group RMT flavors. We also guarantee that each work-item will execute computation in its own SIMD lane, and therefore all SIMD functional units are

protected. Because Intra-Group+LDS doubles the allocation of the LDS and replicates all reads and writes to separate LDS memory locations, the LDS is also protected.

The scalar register file (SRF) and scalar unit (SU) are not included in the SoR. Because the SU exists to compute known shared values for an entire wavefront, if we replicate computation within a wavefront, we will not replicate scalar computation in the SU or registers in the SRF.

Similarly, instruction fetch, decode, and scheduling logic are not protected by Intra-Group RMT. Redundant work-items share this logic because they exist within a single wavefront. Because of the SIMD nature of the architecture, replicating computation among "multiple data" within a wavefront does not protect the "single instruction" state that defines it. Because memory requests from redundant work-item pairs to global memory also can be shared, we conservatively assume that the entire cache hierarchy also must be excluded from the SoR.

6.2. Kernel Modifications

Work-Item ID Modifications: All automatic RMT kernel transformations first rely on doubling the number of work-items (accomplished by the host), and then modifying their ID numbers to create a pair of identical, redundant work-items. Because these ID numbers are the only way work-items can distinguish computation uniquely in OpenCL, work-items that report the same ID numbers, even if they are distinct, will execute the same computation, although with different registers.

To create redundant work-items, we mask the lowest bit of each work-item's global ID. We then save this bit so that work-items can distinguish themselves as either producers or consumers of redundant computation. This also guarantees that redundant work-items will execute in the same wavefront on the GPU, which is important to ensure synchronization. Other OpenCL values, such as a work-item's local ID within a work-group, also require similar transformations.

Including LDS in the SoR: If we choose to include LDS in the SoR (Intra-Group+LDS), we double its allocation and map redundant loads and stores to their own duplicated locations. If we choose to exclude LDS from the SoR (Intra-Group-LDS), we maintain its original allocation, but now must insert output comparisons for every store to the structure.

Communication and Output Comparison: For both Intra-Group+LDS and Intra-Group-LDS, we must allocate a local memory communication buffer in the LDS so that redundant work-items can communicate results. We then identify all instructions where values may exit the SoR and insert code to coordinate output comparisons. For Intra-Group+LDS, we define this to be every operation that potentially modifies off-chip global memory (i.e., all global stores). For Intra-Group-LDS, we also add each local store to this list. Handling of other operations that leave the SoR is left for future work.

Just before a store exits the SoR, the producer work-item is made to communicate both the address and value operands of the store. The consumer work-item then reads these values from the communication buffer and compares them with its private versions. If the values differ, we have detected an error; if identical, the consumer work-item is allowed to execute the store, and redundant computation continues.

6.3. Performance

Figure 2 shows the measured performance overheads for both Intra-Group RMT flavors normalized to the runtime of the original kernel. Because each Intra-Group RMT kernel introduces its own unique pressures on resources within a CU and on the entire GPU, and each RMT flavor reflects those resource pressures in a unique way, we see diverse performance overheads across all kernels.

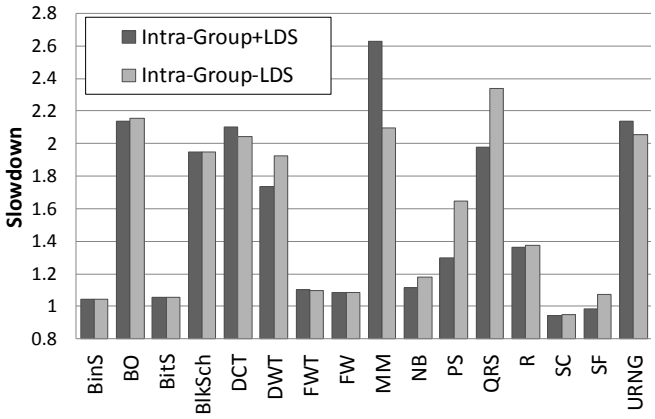

Figure 2: Performance overheads of Intra-Group+LDS and Intra-Group Global-LDS normalized to the runtime of the original kernel. Performance varies wildly depending on the kernel, and RMT flavor.

Intra-Group flavors perform either well, with overheads between 0% and 10%, or poorly, with overheads at or greater than 100%. SimpleConvolution (SC) performs *better* than the original version. This phenomenon seems impossible at first glance, but a few possible reasons are discussed in the following subsection, namely a reduction in resource contention and a reduction in wavefront divergence.

6.4. Analysis

Using performance counters, we were only able to explain some performance behaviors. Therefore, we selectively removed RMT modifications to see if there were any consistent factors that contributed to slow-downs. We analyzed slow-downs caused by communication, redundant computation, and the enhanced size and resource requirements of Intra-Group RMT work-groups. Our findings are discussed below.

Hiding Computation Behind Global Memory Latency: Initially, by analyzing performance counters, we found one main metric–the proportion of time spent executing memory operations relative to vector ALU operations, or "memory boundedness"–explained why some applications did well and

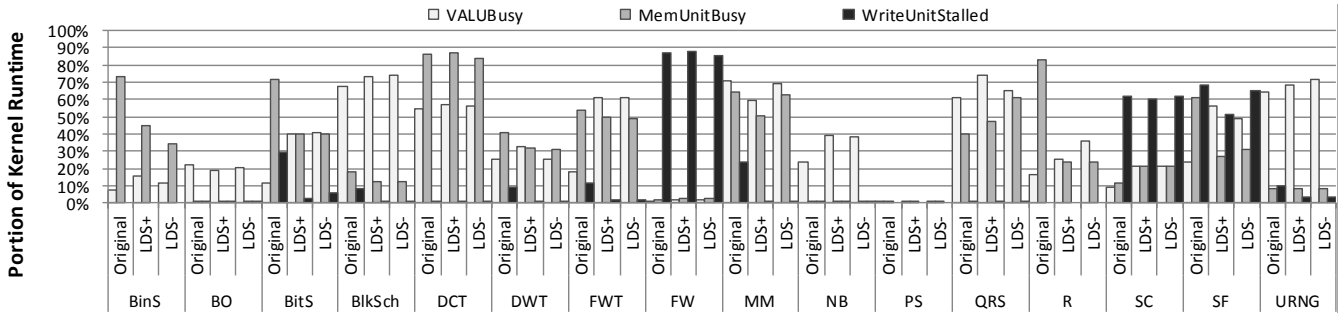

Figure 3: Kernel time spent executing vector ALU operations (VALUBusy) and the proportion of time spent executing memory fetch (MemUnitBusy) and store (WriteUnitStalled) operations. Kernels that have low RMT overheads tend to be memory-bound.

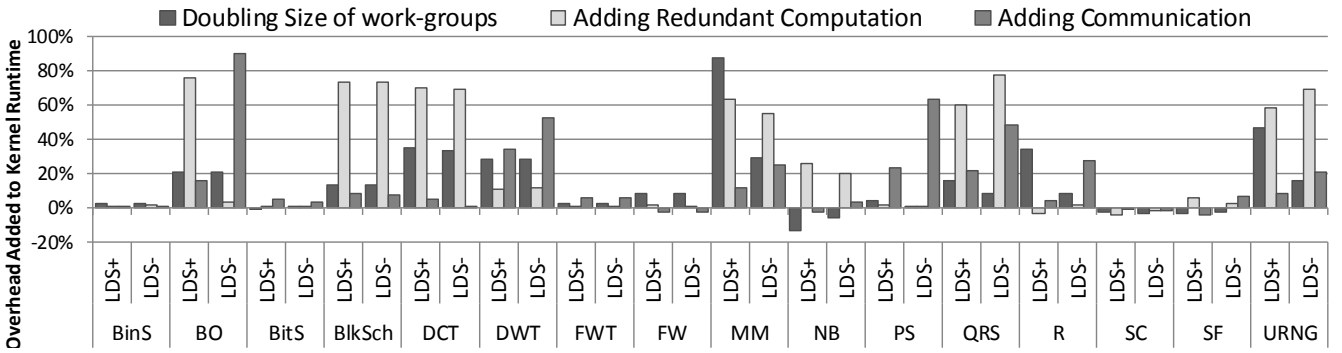

Figure 4: Relative overheads of components of Intra-Group RMT kernels. Each bar represents the additional slow-down added by each successive augmentation to the original kernel. Negative contributions indicate a speed-up over the previous modification.

others did poorly. Figure 3 shows the proportion of time each kernel spent executing vector ALU operations (VALUBusy) or memory operations (MemUnitBusy, WriteUnitStalled). Many kernels that spent most of their time executing global memory operations relative to computation were able to hide the costs of redundant computation and communication, and had relatively low runtime overheads. BinarySearch (BinS), BitonicSort (BitS), FastWalshTransform (FWT), SC, and SobelFilter (SF) all are bound by memory traffic, and have runtime overheads of 10% or less.

In all situations in which Intra-Group RMT overheads were high but kernels spent a large proportion of their time executing memory operations (DCT and MM), kernels also reported spending a large proportion of their time executing vector ALU operations. This indicates that if under-utilization of compute resources exists, software GPU RMT is able to exploit it without hardware modifications. However, if a kernel already heavily utilizes SIMDs or LDS bandwidth, Intra-Group RMT incurs a relatively high overhead regardless of whether the kernel is memory-bound.

Performance counters gave us some evidence to explain the overheads of some kernels, but were not able to account for the overheads of all kernels. For example, DwtHaar1D (DWT) and Reduction (R) are both memory-bound, but do not have low overheads. To gain more insight into performance, we selectively removed three portions of the full RMT transformation (communication, redundant computation, and doubled work-group sizes), isolating each piece's relative contribution to total overhead. Figure 4 shows the relative contributions of each. As suggested by performance counters, DWT and R are able to hide the cost of redundant computation, but must pay a high price for communication and the doubled size of each work-group.

BinomialOption (BO) is a particularly interesting example, because the Intra-Group-LDS version removes the cost of redundant computation, but trades it for an equally large communication penalty. In reality, the runtime of BO is not bound by vector computation or global memory operations, but rather by a high number of local memory accesses. Because Intra-Group-LDS excludes LDS from the SoR, we halve the number of LDS writes per work-group, greatly decreasing the cost of redundant computation. However, because each of these writes exits the SoR, we must insert output comparisons via the LDS for each, proportionately increasing the cost of communication.

Costs of Inter-work-item Communication: By removing the explicit communication and comparison of values between redundant work-items, we were able to measure the relative contribution of inter-work-item communication with respect to the total cost of RMT. Figure 4 shows the results of removing all inter-work-item communication from the Intra-Group RMT algorithms. For a few applications, communication is a very significant portion of the costs of RMT. In BO, DWT, PrefixSum (PS), and R, communication can account for more than half of the total overhead.

In the case of Intra-Group-LDS, communication costs can

be especially expensive for two reasons. First, there simply may be a higher number of output comparisons necessary because we must communicate values on both global and local stores. Second, for local stores, both communication and the protected local store occur via the LDS. Therefore, the cost of communication cannot be hidden behind a higher-latency operation, as is the case in Intra-Group+LDS. Whenever the cost of communication is high relative to the cost of the protected store, communication is expensive.

Costs of Doubling the Size of Work-groups: Typically, Intra-Group RMT causes large scheduling inefficiencies because doubling the size of each work-group will at least halve the number of work-groups that can be scheduled onto any one CU. Additionally, RMT modifications may require the compiler to allocate even more registers than the original kernel, which can cause a further decrease in the number of work-groups that can be scheduled. Because LDS is also a fixed resource per CU, adding the communication buffer or including a work-group's LDS allocation in the SoR (in the case of Intra-Group+LDS), can have similar effects.

To isolate these effects, we artificially inflate the resource usage of the original application to reflect that of the larger work-groups of the RMT version. Results are shown in Figure 4. This modification can be thought of as "reserving" space for redundant computation of larger work-groups without actually executing redundant work-items. We accomplish this by running the original application with the same amount of local memory or VGPRs required by the RMT version, whichever had limited scheduling in the Intra-Group version.

Results indicate that for most benchmarks, when work-group scheduling already is constrained by LDS or VGPR usage, additional resources required by RMT cause a 15-40% overhead. Kernels that suffer from some sort of resource bottleneck mostly suffered from a scarcity of VGPRs, but applications limited by LDS tended to suffer more. For example, LDS over-allocation is responsible for more than half of MM's Intra-Group+LDS RMT overhead, while the Intra-Group-LDS version (limited by VGPR usage) suffers a much smaller penalty. Both Reduction and URNG see similar reductions in scheduling overhead because the Intra-Group-LDS version allocates far less LDS.

Explaining Speed-ups: By doubling the number of wavefronts within a work-group while keeping global memory traffic per work-group approximately the same, Intra-Group flavors essentially halve the memory traffic handled by each CU. This can eliminate bottlenecks in the L1 cache and other structures in the shared memory hierarchy, causing a performance boost. A reduction in contention for both the scalar and vector SIMD units also may contribute to speed-ups of individual groups. This is most likely a consequence of the greedy nature of scheduling that prioritizes high utilization while ignoring runtime contention for resources within a CU. Kernels that saw performance increases after doubling the size of the work-groups, shown in Figure 4, are evidence of this phenomenon.

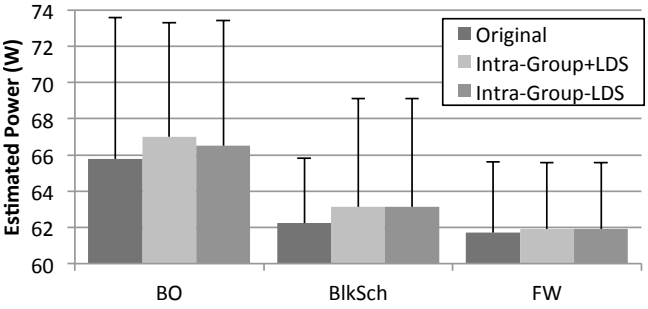

Figure 5: Average power of three SDK workloads with long-running kernels. Peak power is shown as positive whiskers.

A reduction in the penalty for work-item divergence is another possible performance enhancement that may result from Intra-Group RMT flavors, as first documented by [25]. If a single work-item in a wavefront misses in the cache, the entire wavefront must stall until the proper values can be fetched. The instruction scheduler also may issue memory requests from wavefronts that exhibit little locality, causing thrashing in the L1 cache. These issues can be mitigated by shrinking the effective wavefront size so that a cache miss has a smaller penalty. By creating redundant work-item pairs within wavefronts, we accomplish just that. Kernels that saw performance increases after adding redundant computation, shown in Figure 4, are evidence of this phenomenon.

6.5. Power Analysis

Figure 5 shows the average and peak estimated chip power for BO, BlkSch, and FW. These benchmarks were used because they had kernel runtimes long enough to register meaningful measurements. Because the total work-item count in our Inter-Group implementation currently is limited by the size of global memory, we were not able to scale these benchmarks to gather meaningful results for Inter-Group versions of kernels, and so power results are not presented in that evaluation.

Although RMT executes twice as many work-items, all three benchmarks show a small (less than 2%) increase in average power consumption relative to the original application. This is not surprising; if RMT is not taking advantage of under-utilization on the GPU, dynamic power should remain roughly the same. In such cases, the same amount of work is performed with or without RMT, although half of it may be redundant. BlkSch, which for these input sizes has a less than 10% runtime overhead, shows a large peak power increase relative to the original kernel, but on average only increases power consumption by about 1W.

More surprising is FW, which has a negligible power increase but also shows minimal runtime overhead for these larger input sizes, hiding the extra RMT work behind stalls. Our original concern was that taking advantage of under-utilized resources would cause a proportional increase in power consumption, but FW obviously does not follow this logic. Figure 3 shows that although the RMT version of FW

increases the amount of time spent executing ALU operations, the kernel is so memory-bound that this extra time remains a small proportion of overall execution. Therefore, power consumption should not change appreciably. This result is analogous to Amdahl's law: we should expect that even a large increase in power consumption during a small proportion of the computation should not have a large impact on average power.

Although RMT transformations will have a unique effect on power consumption for each benchmark, our preliminary conclusion is that the total energy consumption of each application will be dominated by runtime overheads rather than an increase in power, especially when considering total application runtime and whole-system power.

6.6. Summary

The primary findings from our analysis of Intra-Group RMT are:

- Kernels bottlenecked by global memory operations perform well on Intra-Group RMT because the cost of redundant computation can be hidden behind this latency.
- Communication is expensive whenever the cost of communication is high relative to the cost of the protected store because the access to the protected store cannot mask the latency of communication.
- RMT exacerbates pressure on shared resources such as VGPRs and LDS. Therefore, kernels that are bottlenecked on shared resources will see high overheads due to RMT.
- Intra-Group RMT can cause accidental performance improvements by reducing wavefront divergence, cache thrashing, and over-utilization of other shared CU resources.
- Average power consumption is not affected appreciably by Intra-Group RMT; therefore, total energy consumption will be proportional to the runtime overhead in each kernel.

Intra-Group RMT overheads vary highly depending on kernel behavior and are difficult to predict. Therefore, automatic RMT should be applied in development to help the programmer further tune software for the best RMT performance. Programs should be tuned to avoid bottlenecks on shared computation resources. Furthermore, our study suggests that RMT performance could be improved by more efficient register allocation in the compiler, and by allowing hardware or the compiler to adjust wavefront size for a given application.

7. Inter-Group RMT

Inter-Group RMT replicates work by doubling the number of work-groups in the global NDRange and then assigning redundant work-item pairs in separate work-groups based on OpenCL's work-item identification numbers. Because OpenCL prohibits work-items in different work-groups from sharing the same LDS address space, we must implement output comparisons in the global memory hierarchy.

7.1. Sphere of Replication

Table 3 shows the structures that are included in the SoR for Inter-Group RMT. Because redundant threads execute within separate work-groups, we know that all scalar instructions will be duplicated, and allocated their own set of scalar registers. Unlike Intra-Group RMT, we therefore can consider the entire SU, including the SRF, inside the SoR. We also are guaranteed that vector instructions will be duplicated because redundant threads are guaranteed to be in different wavefronts. Therefore, the instruction fetch, scheduling, and decode logic of each CU can be considered inside of the SoR, as well as the VRF and SIMD units. We can also include the LDS in the SoR because each work-group is given its own allocation. Similar to the SRF and VRF, LDS is divided among groups until it, or the SRF or VRF, becomes a limiting resource.

	SIMD ALU	VRF	LDS	SU	SRF	ID	IF/SCHED	R/W L1$
Inter-Group	✓	✓	✓	✓	✓	✓	✓	

Table 3: CU structures protected by Inter-Group RMT. Because redundant work-items exist in separate wavefronts, most structures on the CU exist within the SoR.

While these structures are guaranteed to be inside the SoR, memory requests to the R/W L1 cache still may be unprotected by Inter-Group RMT because two groups can be scheduled on the same CU and they may or may not share the same L1 cache. If this occurs, requests from both groups can be fulfilled from the same line in the L1. Because we cannot eliminate this situation, we conservatively assume that the L1 cache is outside the SoR.

7.2. Kernel Modifications

Adding Explicit Synchronization: For Intra-Group RMT, we can guarantee ordering of communication because redundant work-items executed in lockstep within the same wavefront. However, in OpenCL, no global synchronization is guaranteed between work-items in different work-groups. Therefore, we must introduce explicit synchronization to coordinate communication between work-items.

Work-item ID Modifications: Because we never can guarantee scheduling of an equal number of producer and consumer groups, we may end up introducing deadlock if we rely on a work-group's given ID for producer/consumer distinction. For example, if all scheduled groups are consumers, they will starve while waiting for producer groups to communicate redundant values. To avoid deadlock, a work-item from each newly executing work-group acquires and increments a global counter. This counter acts as a new work-group ID, allowing us to control work-group producer/consumer assignment to avoid deadlock.

Similar to the Intra-Group RMT algorithms, each work-item also needs to save the lowest bit of the acquired work-group ID (the producer/consumer distinguishing flag) and then mask the lowest bit to create identical redundant work-group pairs. This flag determines which work-groups within a redundant

pair will be a producer or consumer of redundant values for output comparison.

Once all work-items have acquired their new work-group IDs, they need to create new global work-item ID numbers to match up with the acquired work-group ID's position in the global NDRange.

Unlike Intra-Group RMT, we do not need to remap any local ID numbers or group size queries because each local NDRange is identical in size and dimensionality to the original computation. Also, because each work-item calculates a new global ID, we do not need to adjust existing global ID queries; instead, we replace all of their uses with the new calculated work-item ID values.

Coordinating Communication and Output Comparisons: Inter-Group RMT communication between work-items is more expensive than Intra-Group RMT communication between work-items. Because separate work-groups cannot share a local address space, we allocate communication buffers in global memory, which greatly increases inter-work-item communication latency. After each group is set up for redundant execution, we identify every place the data exits the SoR and insert output comparison code. For Inter-Group RMT, data exits the SoR whenever any operation accesses the global memory hierarchy.

To synchronize output comparisons between between redundant work-items in different work-groups, we use a two-tiered locking system. First, a work-item in the producer work-group that wants to store out to global memory waits for a particular communication buffer to become available. Once a buffer has been acquired, that work-item stores its potentially incorrect values on the buffer and signals the work-item in the consumer work-group to read these values. Once the consumer work-item reads this signal, it reads the values from the communication buffer and compares them with its private values. After the consumer work-item verifies that the values are identical, it executes the store and then frees the communication buffer for use by another producer work-item.

Any communication using global memory is complicated by the cache hierarchy and the relaxed-consistency memory model present on many modern GPUs. AMD GPU L1 caches are write-through, making all writes globally visible in the L2, but not necessarily visible in the L1 caches of other CUs. To solve this issue, we force work-items to read from the L2 cache by issuing atomic adds with the constant 0. This ensures the read will be from the backing store and that the most recent, globally visible value at that address will be returned.

7.3. Performance

Figure 6 shows that Inter-Group RMT has an even wider range of performance than Intra-Group RMT, from SC, which has an average slow-down of 1.10x, to BitS, which has an average slow-down of 9.48x. Performance counters gave no indication as to why RMT became so expensive for some applications (mainly BitS, DWT, and FWT).

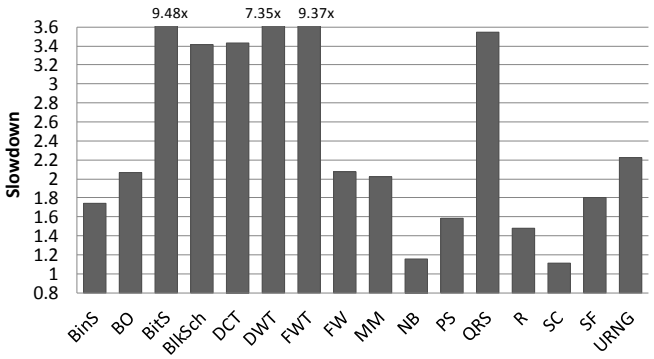

Figure 6: Performance overheads of Inter-Group RMT normalized to the runtime of the original kernel. Some kernels have extremely large slow-downs due to the high cost of communication.

7.4. Performance Analysis

To identify the causes of poor performance, we selectively removed portions of the RMT algorithm to isolate for their effects. Figure 7 shows the results of our experiments. We first removed communication code from the RMT algorithm to measure the cost of inter-work-item communication. We then modified each group of the original kernel to use the same number of resources as two Inter-Group RMT groups. This modification approximates the cost of scheduling twice as many work-groups on the GPU without adding extra computation or memory traffic. The remaining overhead of Inter-Group RMT is the cost of actually executing redundant work-groups and synchronizing their global memory accesses.

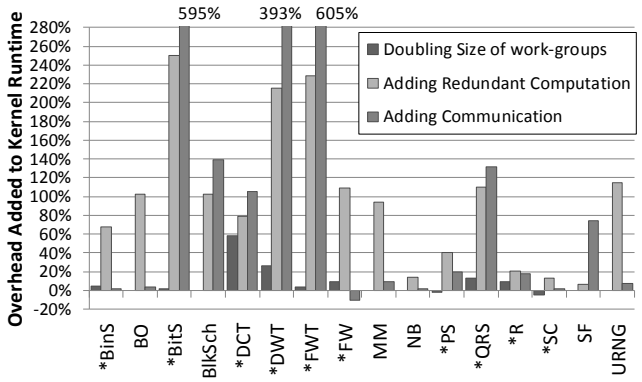

Figure 7: Each bar represents the additional slow-down added by each successive augmentation to the original kernel. Negative contributions indicate a speed-up relative to the previous modification. Kernels marked with * indicate that the work-group doubling experiment was performed.

Costs of Inter-work-item Communication: Because Inter-Group RMT uses global memory for inter-work-item communication, we would expect this cost to be extremely high. However, Figure 7 shows that for most kernels, communication is not the main bottleneck; in fact, communication is a very small proportion of the overall overhead of Inter-Group RMT. This is a counter-intuitive result, but can be explained

when considering the pre-existing bottlenecks of each kernel.

The number of required output comparisons that Inter-Group RMT requires scales with the total number of writes to global memory. Therefore, for applications that have a low proportion of global writes with respect to kernel computation, the cost of communication will be low. In BinS, many work-itemss never execute a global write at all and never require synchronization or communication.

In all of the applications that do extremely poorly (greater than 3x slow-down), communication is a large contributing factor. The reason for such a high overhead relative to other applications turns out to be pre-existing bottlenecks in the memory hierarchy. If a kernel spends a high proportion of time reading, writing, or stalling on memory operations, adding any extra reads, writes, and atomics can be especially expensive. For example, the original BitS kernel spent about 72% of its execution time issuing memory commands and 26% of its time executing writes. After adding the additional memory traffic for communication and synchronization, BitS spent almost 94% of its time executing memory operations, and 59% of its total execution time stalled.

High CU utilization also may amplify this effect. Because wavefronts executing on a CU share the L1 cache, if we increase the number of memory operations per CU, we may introduce memory stalls or cause thrashing in the cache. All applications with low communication overheads also spent a low proportion of execution time doing memory operations, wrote a small number of bytes in total to memory, and had a low proportion of stalls in the memory hierarchy.

CU Under-utilization: One candidate to explain low overheads of Inter-Group RMT is CU under-utilization. If an application does not launch enough work-groups to saturate all CUs on the chip, there will be little or no added contention for CUs. When developing GPU programs for massively parallel algorithms, it generally is easy to launch enough work-groups to utilize all CUs; therefore, too few work-groups should not be considered a realistic situation. For the input sizes used for testing, the applications caused only two kernels to under-utilize CUs on the GPU: NBody (NB) and PS, which utilize only eight and one of twelve available CUs, respectively. Both perform well under Inter-Group RMT transformations, with overheads of 1.16x and 1.59x, respectively. However, this does not explain why applications that fully utilize CUs (BinS, SC, R, and SF) also do well.

Explaining Less Than 2x Slow-downs: While kernels that saturate the LDS and compute capabilities of a CU experience expected 2x overheads after adding redundant computation (BO, BlkSch, DCT, FW, MM, QRS, URNG), other kernels experience less than 2x overheads. Kernels like BinS and R are designed such that not all groups that are scheduled to the GPU write back to global memory. Therefore, these "ghost" groups never need to communicate values for output comparisons. If a kernel is already memory-bound, the computation of these ghost groups, and their redundant partners, may be hidden behind other global memory traffic.

SC and SF may benefit from another accidental optimization called "slipstreaming" [22]. Slipstreaming is a form of pre-fetching in which a previous speculative work-item warms up caches for more efficient execution. Because SC and SF are both image transformations that modify pixels based on surrounding pixel values, work-groups in these kernels share a large amount of reads. Redundant groups in kernels with such behavior therefore may benefit from a form of slipstreaming because redundant groups prefetch data into the cache hierarchy.

Costs of Doubling the Number of Work-groups: Figure 7 shows the cost of doubling the number of work-groups. To approximate this cost, we inflated the number of VGPRs required by the original application to match that required by two Inter-Group RMT work-groups. For applications whose Inter-Group RMT versions executed an odd number of work-groups per CU, this technique could not match the work-group occupancy required to simulate Inter-Group RMT occupancy, and so we present results for only a subset of the applications.

Only one application, DCT, was affected significantly by doubling the number of work-groups. This indicates that for most kernels, scheduling more groups is not a major bottleneck. SC was the only kernel that saw an appreciable speed-up due to a decrease in utilization and contention among shared CU resources, an effect discussed in previous sections.

7.5. Summary

The primary findings from our analysis of Inter-Group RMT are:

- Additional memory traffic caused by communication and synchronization is extremely expensive if the application already is bottlenecked by global memory accesses, cache thrashing, or memory stalls.
- CU under-utilization can allow for relatively low overhead Inter-Group RMT but is not a realistic scenario for many production applications.
- If a kernel is compute or LDS throughput-bound, Inter-Group RMT shows an expected 2x slow-down.
- Kernels that experience slow-downs less than 2x may be benefiting from "slipstreaming" [22], or from groups that never write to global memory, which therefore never need to communicate and can hide their computation behind global memory latency.

GPUs are designed to run efficiently as many work-groups as possible in parallel, which typically is achieved by not having any ordering constraints or communication among groups. Because group ordering and inter-group communication is an inherent feature of Inter-Group RMT, kernel performance can degrade. Alternative approaches that facilitate more efficient inter-group communication in GPUs, such as task queuing (e.g., [13, 31]) or co-scheduling techniques, may alleviate this bottleneck.

8. Going Beyond OpenCL: Leveraging Architecture-specific Features

To maintain portability across different architectures, we originally chose to implement our RMT transformations within the constraints of the OpenCL specification. However, OpenCL imposes certain restrictions on how work-items are organized and executed, and how they communicate with each other, on the GPU. Even if individual GPU or CPU architectures allow for more general behaviors, their OpenCL implementations may hide these capabilities.

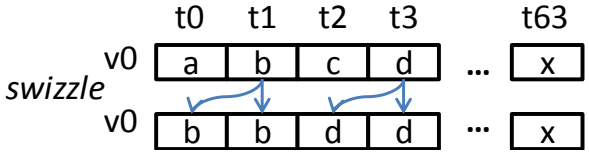

Figure 8: The behavior of the swizzle instruction, used for inter-work-item communication. Values in V0 from odd lanes (t1, t3, etc.) are duplicated into even lanes (e.g., after the swizzle instruction, work-item 1s value b now can be read by work-item 0).

By taking advantage of architecture-specific capabilities, we may be able to reduce the overheads of RMT without affecting the correctness or portability of the original application. While these capabilities may improve performance, they are not portable, but could be disabled on architectures that do not support them.

One example of an AMD GPU-specific capability not exposed by OpenCL is the swizzle [6] instruction. A swizzle allows the re-organization of 32-bit values within a vector register. OpenCL allows intra-group, inter-work-item communication only via local memory. However, because we can guarantee that Intra-Group RMT work-item pairs execute within a single wavefront, and therefore share vector registers, we can use the swizzle instruction to share values directly via the VRF. Implementing this fast inter-work-item communication may result in lower latency as well as reduced resource pressure by eliminating the need for a local memory communication buffer.

Figure 8 shows an example of how communication can occur using the swizzle instruction in a wavefront's vector register. Registers in the VRF are organized into 64, 32-bit lanes in which each work-item within a wavefront reads and writes values to single lane. The swizzle instruction allows a wavefront to re-organize the values into different lanes, therefore providing a way to pass private values between work-items without the need for local memory.

By modifying the low-level GPU shader compiler to perform inter-work-item communication via the swizzle instruction, we saw significant speed-ups for kernels with high communication costs. The results are shown in Figure 9. BO, DWT, PS, and QRS all see considerable improvements in performance. FW and NB show decreases in performance, possibly because of the overhead of added casting and packing/unpacking of vectors necessary for communication via the 32-bit registers.

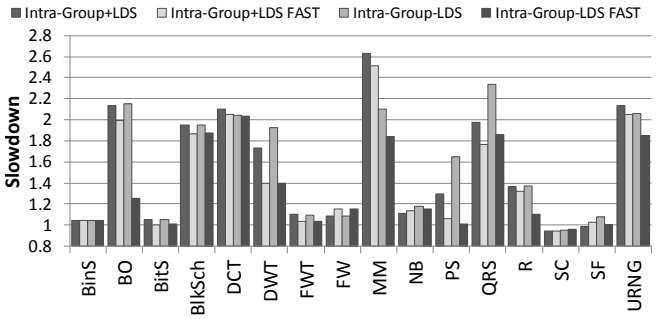

Figure 9: Intra-Group RMT results before and after implementing fast inter-work-item communication via the VRF.

This example shows that if we allow our implementation to go beyond the OpenCL specification, we can realize significant performance improvements in some cases. This motivates further research into hardware and compiler framework modifications that further decrease the overheads of RMT.

9. Conclusions and Future Work

In this work, we evaluated performance and power overheads of three automatic techniques for software-implemented redundant multithreading on graphics processing units targeting different spheres of replication. The performance of GPU RMT depends on the unique behaviors of each kernel and the required SoR, but generally favors kernels that under-utilize resources within compute units. Some kernels are even accelerated by RMT transformations due to accidental optimizations and the relief of some fixed, per-CU resource bottlenecks.

Increasing the required SoR generally involved a performance trade-off. The larger the granularity of replication in OpenCL, the larger the SoR. However, this increase in protection from transient faults often resulted in a reduced ability to take advantage of under-utilization on the chip as well as higher inter-work-item communication and synchronization costs.

We evaluated a hardware mechanism to reduce communication overheads by implementing fast inter-work-item communication in the vector register file rather than local memory. This change improved performance in kernels with large communication overheads; however, such a solution is not exposed in the OpenCL specification, motivating further research into architecture-specific capabilities that may help decrease the overheads of RMT.

10. Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank John Kalamatianos, Daniel Lowell, Mark Wilkening, Bolek Ciesielski, and Tony Tye for for their valuable inputs.

References

[1] LLVM. [Online]. Available: http://llvm.org

[2] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang,

K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu, *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*, 2011.

[3] AMD. AMD CodeXL. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/

[4] AMD. AMD Graphics Cores Next (GCN) Architecture. Available: http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf

[5] AMD. OpenCL Accelerated Parallel Processing (APP) SDK. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/downloads/

[6] AMD. (2012) Southern Islands Series Instruction Set Architecture. Available: http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

[7] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 58:1–58:11. Available: http://dl.acm.org/citation.cfm?id=2388996.2389075

[8] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, Jul. 2003. Available: http://dx.doi.org/10.1109/MM.2003.1225959

[9] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 94–104. Available: http://doi.acm.org/10.1145/1513895.1513907

[10] D. Foley, M. Steinman, A. Branover, G. Smaus, A. Asaro, S. Punyamurtula, and L. Bajic, "AMD's Llano Fusion APU," in *"IEEE/ACM Symposium on High Performance Chips (HOTCHIPS)"*, 2011. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.930-Llano-Fusion-Foley-AMD.pdf

[11] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 172–183. Available: http://dx.doi.org/10.1109/ISCA.2005.38

[12] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 691–696. Available: http://dx.doi.org/10.1109/CCGRID.2010.84

[13] HSA Foundation. Heterogeneous System Architecture Specification. Available: http://hsafoundation.com/standards/

[14] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1056–1057. Available: http://dx.doi.org/10.1109/DATE.2005.98

[15] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 37–47. Available: http://dx.doi.org/10.1109/MICRO.2012.13

[16] D. Kinniment, I. L. Sayers, and E. G. Chester, "Design of a reliable and self-testing VLSI datapath using residue coding techniques," *Computers and Digital Techniques, IEE Proceedings E*, vol. 133, no. 3, pp. 169–179, 1986.

[17] M. Lovellette, K. Wood, D. L. Wood, J. Beall, P. Shirvani, N. Oh, and E. McCluskey, "Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed," in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 5, 2002, pp. 5–2109–5–2119 vol.5.

[18] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 99–110. Available: http://dl.acm.org/citation.cfm?id=545215.545227

[19] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[20] R. Nathan and D. J. Sorin, "Argus-D: A Low-Cost Error Detection Scheme for GPGPUs," ser. Workshop on Resilient Architectures (WRA), Atlanta, GA, 2010.

[21] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[22] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "Slipstream Memory Hierarchies," Tech. Rep., 2002.

[23] M. K. Qureshi, O. Mutlu, and Y. N. Patt, "Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors," in *In Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005*, 2005, pp. 434–443.

[24] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 25–36, May 2000. Available: http://doi.acm.org/10.1145/342001.339652

[25] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 72–83. Available: http://dx.doi.org/10.1109/MICRO.2012.16

[26] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware," in *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '06. New York, NY, USA: ACM, 2006, pp. 9–16. Available: http://doi.acm.org/10.1145/1283900.1283902

[27] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 389–398. Available: http://dl.acm.org/citation.cfm?id=647883.738394

[28] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, June 2007, pp. 297–306.

[29] J. Tan and X. Fu, "Rise: improving the streaming processors reliability against soft errors in gpgpus," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 191–200. Available: http://doi.acm.org/10.1145/2370816.2370846

[30] The Khronos Group. The OpenCL Specification. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[31] S. Tzeng, B. Lloyd, and J. D. Owens, "A GPU Task-Parallel Model with Dependency Resolution," *Computer*, vol. 45, no. 8, pp. 34–41, 2012.

[32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th Annual International Symposium on Computer architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 87–98. Available: http://dl.acm.org/citation.cfm?id=545215.545226

[33] M. Villmow. AMD OpenCL Compiler. Available: http://llvm.org/devmtg/2010-11/Villmow-OpenCL.pdf

[34] C. Wang, H.-s. Kim, Y. Wu, and V. Ying, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 244–258. Available: http://dx.doi.org/10.1109/CGO.2007.7

[35] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 287–300. Available: http://dx.doi.org/10.1109/IPDPS.2011.36

[36] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime asynchronous fault tolerance via speculation," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 145–154. Available: http://doi.acm.org/10.1145/2259016.2259035